PureBasic
Ein WikiBooks-Projekt

Karsten
Sulfur, JonRe

Inhaltsverzeichnis

1 Einstieg

1.1 Über PureBasic

PureBasic (PB) ist ein proprietärer Basic-Dialekt, der von Frédéric Laboureur im Jahre 2000 aus dem Beta-Status entlassen wurde. Die Sprache ging aus Bibliotheken hervor, die Laboreur in den 1990er Jahren für BlitzBasic auf dem Amiga entwickelte. Heute erhält man neben dem Compiler noch eine IDE und viele andere Werkzeuge, die das Programmieren vereinfachen. Eine beschränkte Demo-Version ist ebenfalls verfügbar.

PB ist für Windows, Linux und Mac OSX verfügbar und wird aktiv weiterentwickelt. Durch die klassisch-einfache Basic-Syntax ist PB einfach zu erlernen und verfügt trotzdem über Features, die für die fortgeschrittene und profesionelle Programmierung notwendig sind, wie z.B. Zeiger. Des Weiteren bietet PB viele Sprachelemente, um auf die API des jeweiligen Systems zuzugreifen, wodurch z.B. die GUI-Entwicklung vereinfacht wird. Inline-Assembler wird ebenfalls unterstützt. Weitere Features sollen hier dargestellt werden:

- Nutzung von Medien (z.B. CDs, MP3, AVI)
- Bibliothek für Sprites
- Integrierte 3D Engine (OGRE)
- Unterstützung von DLLs
- Verarbeitung von XML Dokumenten
- Bibliotheken für Netzwerkprogrammierung

Seit 2006 ist außerdem objektorientiertes Programmieren als Plugin verfügbar.

2 Grundlagen

2.1 Hello, world!

Die Tradition des „Hello, world!“-Programmes wurde von Brian Kernighan eingeführt, als er eine Dokumentation über die Programmiersprache B schrieb. Durch das Buch *The C Programming Language* erlang das Beispiel Bekanntheit.

"Hello, world!ïst hervorragendes Beispiel für jede Programmiersprache, da man mit jenem die wesentlichen Aspekte dieser erfassen und darlegen kann. Von daher ist es nur angebracht es hier ebenfalls zu verwenden.

```
; Listing 1: Hello, world!

OpenConsole()
PrintN("Hello, world!")
Delay(2000)

Ausgabe:

Hello, world!
```

Nachdem man PureBasic installiert hat, öffnet man die IDE über das Startmenü (Windows) bzw. über den Befehl *purebasic* (Linux & Mac OSX). Eine IDE ist eine Entwicklungsumgebung, in der alle Werkzeuge, die man zum Programmieren benötigt, organisiert und schnell verfügbar sind. Man gibt nun den oberen Codeabschnitt ein und drückt F5. Für dieses und alle anderen Kapitel soll gelten, dass man die Programme am Besten von Hand eingibt, da man durch simples Copy&Paste keine Programmiersprache lernt, sondern indem man selber schreibt und ausprobiert!
Hat man die Anleitung beachtet, müsste nun ein Fenster offen sein, in dem *Hello, world!* steht und das sich nach 2 Sekunden wieder schließt. Unter Linux wird die Ausgabe direkt auf der Shell sichtbar, über die man die IDE öffnete. Warum geschieht dies? Die erste Zeile stellt einen sogenannten Kommentar dar, d.h. eine Befehlszeile, die von sogenannten *Compiler* ignoriert wird. Dies sieht man an dem Semikolon, dass am Anfang der Zeile steht. Kommentare können überall im Code stehen, auch hinter einer Codezeile. Der Compiler ist das Programm, dass den für den Programmierer lesbaren Code in eine für den Computer ausführbare Datei übersetzt.
Danach kommt der erste Befehl: *OpenConsole()*. Dieser tut nichts anderes, als unter Windows eine Konsole zu öffnen, unter Linux wird er ignoriert, da Konsolenprogramme, wie dieses eines ist hier immer über die Shell gestartet werden.
Es werden hier wesentliche Eigenschaften von PureBasic klar: Der PureBasic-Compiler

arbeitet zeilenorientiert, d.h. pro Zeile steht ein Befehl. Des Weiteren ist in PureBasic die Groß- bzw. Kleinschreibung zu beachten. *openConsole()* würde also nicht funktionieren, da das ö"groß geschrieben werden müsste. Zuletzt muss hinter jedem Befehl ein Klammerpaar stehen. In diesem stehen die Argumente für den Befehl. Was dies ist wird beim nächsten Befehl erklärt.
PrintN("Hello, world!") tut nichts anderes, als die Zeile *Hello, world!* auf die Ausgabe zu schreiben und danach einen Zeilenumbruch einzufügen. *"Hello, world!"* ist hierbei das Argument von *PrintN()*, also etwas, das von *PrintN()* zur Ausführung benötigt wird, in diesem Fall wird ein String benötigt. Was das ist, wird in einem anderen Kapitel erklärt. *Delay(2000)* hält die Ausführung für 2000 Millisekunden, also 2 Sekunden, an. 2000 ist wieder ein Argument, diesmal jedoch ein Zahlenwert. Es fällt auf, dass das Argument von *Delay()* nicht in Anführungsstrichen steht. Dies liegt daran, dass *Delay()* einen Zahlenwert erwartet und keinen String wie *PrintN()*, denn Strings stehen in PureBasic immer in Anführungsstrichen, ansonsten werden sie vom Compiler nicht als solche interpretiert.

Nachdem nun klar ist, wie das Programm funktioniert, sollte man ein bisschen mit diesem herumspielen, um sich mit der Syntax vertraut zu machen. Was passiert z.B., wenn man die Anführungstriche bei *PrintN("Hello, world!")* weglässt? Man sollte sich mit den Ausgaben des Debuggers vertraut machen; diese helfen weiter, wenn etwas falsch programmiert wurde oder andere Probleme auftreten.

Wenn man Fragen zu einem Befehl hat, kann man auf diesen klicken und dann F1 drücken. So wird direkt die Hilfeseite zu diesem Befehl geöffnet. Ansonsten gelangt man über F1 auch in die allgemeine Referenz, die auch im Internet eingesehen werden kann. Der Links ist im Inhaltsverzeichnis verfügbar.

2.1.1 Der Debugger

Der Debugger zeigt, wie schon gesagt, Fehler an. Er wird durch das Fenster unter dem Editor repräsentiert. Lässt man z.B die Anführungszeichen weg, zeigt er beim kompilieren an: "Line 4: Syntax Error."Mit dieser Information kann man viel anfangen, denn man weiß nun, dass das Programm nicht kompiliert, weil in Zeile 4 etwas falsch ist und ausgebessert werden muss.

Der Debugger kann auch in der Entwicklung eines Projekts nützlich sein, denn mit ihm kann man die Ausführung des Programmes dokumentieren.

```
; Listing 2: Hello, world! mit Debugger

OpenConsole()
Debug "Jetzt kommt die Ausgabe"
PrintN("Hello, world!")
Debug "Nun wartet das Programm 2 Sekunden"
Delay(2000)

Ausgabe:
```

```
Hello , world!
```

Man drückt abermals auf F5, nachdem man das Programm eingegeben hat. Es kommt wieder die Ausgabe aus dem ersten Programm, jedoch passiert noch etwas anderes: Es öffnet sich ein Fenster in dem die Strings hinter *Debug* angezeigt werden. Die Strings sind hierbei aber keine Argumente, sondern sogenannte Ausdrücke, wie z.B. *PrintN("Hello, world!")*, also etwas, das ausgewertet werden kann. Es gibt noch viele weitere solcher Debugger-Schlüsselwörter. Das Wort SSchlüsselwörter"wird in einem anderen Kapitel erläutert.
Das Interessante an diesen Schlüsselwörtern ist, dass sie nur bei aktivierten Debugger kompiliert werden. Dies ist standardmäßig der Fall. Sie eignen sich also eine großartige Hilfestellung bei der Projekterstellung, ohne dass man sich in der finalen Version darum kümmern muss alle Befehle zu entfernen.

Der Debugger kann über den Reiter *Debugger* deaktiviert werden, in der Demo-Version lässt er sich jedoch nicht ausschalten.

2.1.2 Aufgaben

1. Niemals vergessen: Programmieren lernt man nur durch selber schreiben und ausprobieren, also auch durch bewusstes Fehler einbauen und verändern von Code.

2. Was ist an folgendem Programm falsch?

   ```
   OpenConsole() PrintN(Hello , world!)
   Delay(2000
   ```

2.2 Variablen und Konstanten

Variablen sind einfach gesagt Datenspeicher. In ihnen können Zahlenwerte, als auch Strings zur späteren Benutzung gespeichert werden.

```
; Listing 3: Variablen

OpenConsole()

x = 1
PrintN(Str(x))

x = 2
PrintN(Str(x))

y = x+1
PrintN(Str(y))

y + 1
```

```
PrintN(Str(y))

Delay(2000)

Ausgabe:

1
2
3
4
```

Nachdem das Programm wie üblich kompiliert wurde, sieht man die Zahlen 1, 2, 3 und 4 untereinander ausgegeben. Neu sind die Variablen.
Variablen funktionieren an sich ganz einfach: Man *deklariert* und *definiert* sie, d.h. man teilt dem Compiler mit das es nun eine neue Variable mit einem Bezeichner gibt und ihr wird automatisch vom Compiler Speicherplatz zugewiesen. Das funktioniert an jeder Stelle im Code. Danach hat die Variable jedoch noch keinen Wert. Dies tut man über die *Initialisierung*. In PureBasic kann man alle drei Schritte gleichzeitig durchführen bzw. alle zwei, denn Deklaration und Definiton geschehen immer zusammen. *x = 1* deklariert und definiert also die Variable mit dem Bezeichner x und initialisiert sie mit dem Wert 1. Es ist zu beachten, dass Variablenbezeichner nicht mit Zahlen beginnen dürfen und keine Rechenoperatoren oder Sonderzeichen enthalten dürfen. Dazu zählen auch Umlaute und ähnliche Zeichen.
PrintN(Str(x)) gibt den Inhalt der Variable x, also 1, aus. Warum jedoch muss man schreiben *Str(x)*? Wenn man sich an Listing 1 erinnert, wurde dazu erklärt, dass *PrintN()* immer einen String als Argument erwartet, jedoch keine Zahl, wie z.B. *Delay()*. Deshalb muss x zuvor in einen String umgewandelt werden, was *Str()* erledigt. Es fällt auf, dass PureBasic innere Ausdrücke vor äußeren auswertet (*Str()* vor *PrintN()*), genauso wie man es in der Grundschulmathematik bei der Klammersetzung gelernt hat.
Des Weiteren kann man Variablen einen neuen Wert zuweisen. Dies geschieht auf die gleiche Weise wie die Initialisierung. Am Beispiel von y sieht man außerdem, dass mit Variablen gerechnet werden kann.

In PureBasic sind alle Standardrechenmethoden verfügbar:

- '+' für Addition
- '-' für Substraktion
- '*' für Multiplikation
- '/' für Division

2.2.1 Konstanten

Wie man sah, können Variablen dynamisch Werte zugewiesen werden. Es gibt aber auch Fälle, in denen man eine Konstante haben will, also einen Bezeichner für einen Wert,

der sich nicht ändert.

```
; Listing 4: Konstanten

#Vier = 4

PrintN(Str(#Vier))
Delay(2000)
```

Ausgabe:

4

In diesem Fall wurde die Konstante #Vier erzeugt (die Raute gehört zum Bezeichner!). Sie kann nachträglich nicht mehr geändert werden und repräsentiert in nachfolgenden Code den Zahlenwert 4. Ein nachträgliches *#Vier = 5* würde also einen Compilerfehler erzeugen.

Es ist zwar prinzipiell freigestellt, welche Bezeichner man für Konstanten und Variablen wählt, jedoch gilt die Konvention, dass Konstanten groß geschrieben werden und Variablen klein. Dies dient der besseren Unterscheidung und macht den Code insgesamt lesbarer.

PureBasic verfügt über viele interne Konstanten, z.B. *#Pi* für die Kreiszahl. Andere stehen in der PureBasic-Referenz, wobei die meisten zu diesem Zeitpunkt noch verwirren können, da sie für verschiedene Befehle als Optionen zur Verfügung stehen oder zur Verarbeitung dieser.

2.2.2 Eingabe

Natürlich möchte man auch die Möglichkeit haben Variablenwerte vom Benutzer des Programmes festlegen zu lassen. Man denke z.B. an einen Taschenrechner.

```
; Listing 5: Benutzereingabe

OpenConsole()

Print("1. Zahl: ")
x = Val(Input())
Print("2. Zahl: ")
y = Val(Input())

PrintN("x + y = "+Str(x+y))

Delay(2000)
```

Ausgabe:

```
1. Zahl: 1
2. Zahl: 3
x + y = 4
```

In diesem Beispiel wurden wieder viele neue Dinge eingeführt. Was sieht man bei der Ausführung? Das Programm gibt den ersten String 1. Zahl: aus und wartet dann darauf das eine Zahl vom Benutzer eingegeben wird. Danach muss eine weitere Zahl eingegeben werden und zuletzt wird die Summe aus beiden ausgegeben.
Zuerst fällt auf, dass diesmal *Print()* und nicht *PrintN()* benutzt wird. Der Unterschied zwischen beiden liegt einzig und allein darin, dass *Print()* keinen Zeilenumbruch erzeugt, also die nächste Ausgabe auf die gleiche Zeile ausgegeben wird.
Als nächstes ist der Befehl *Input()* neu. Er wartet bis der Benutzer etwas eingibt und Enter drückt. Die Eingabe wird als String zurückgegeben, das bedeutet man erhält von dem Befehl etwas (in diesem Fall einen String), mit dem man weiterarbeiten kann. Außerdem erzeugt *Input()* einen Zeilenumbruch nach der Eingabe. Da wir jedoch einen Zahlenwert speichern wollen und keinen String, muss dieser nun in einen Zahlenwert umgewandelt werden. Dies geschieht mit dem Befehl *Val()*, der als Gegenstück zu *Str()* gesehen werden kann.
Zuletzt fällt auf, dass die Zahlenoperation, das Addieren, direkt in den *Str()*-Befehl geschrieben wurde. Es wird also zuerst der Ausdruck in den Klammer ausgewertet, bevor der Befehl an sich ausgewertet wird. Dies gilt auch für alle anderen Befehle. Des Weiteren kann man Strings zusammenfügen, ebenfalls über +.

2.3 Variablentypen

Es existieren mehrere sogenannter Variablentypen. Jede Variable hat einen ihr zugewiesenen Typ, wobei der voreingestelle *Long* ist, was bedeutet, dass die Variable eine Ganzzahl ist und im Speicher 4 Byte einnimmt. Die Größe im Arbeitsspeicher ist entscheidend für die maximale Größe des Wertes. Im Arbeitsspeicher sind alle Zahlen im binären System abgespeichert, d.h. als Nullen und Einsen. Ein Byte besteht aus 8 Bits, also 8 Stellen im binären System. Wenn nun ein Long 4 Byte groß ist, können in ihm Zahlen abgespeichert werden, die im binären System insgesamt 32 Bits groß sind. Da in PureBasic Variablen bis auf wenige Ausnahmen ein Vorzeichen erhalten, halbiert sich die maximale Größe des Wertes noch einmal.

Die wichtigsten Variablentypen sind:

- Long, für Ganzzahlen im Bereich -2147483648 bis +2147483647
- Float, für Gleitkommazahlen (unbegrenze Größe)
- Quad, für große Ganzzahlen von -9223372036854775808 bis +9223372036854775807 (8 Byte)
- Ascii bzw. Unicode, für ein Zeichen (1 bzw. 2 Byte)

- String für Zeichenketten (Länge des Strings + 1)
- Andere Variablentypen stehen in der Referenz

Im Quellcode kann festgelegt werden, von welchem Typ die Variable ist.

```
; Listing 6: Variablentypen

OpenConsole()

x.l = 5
y.f = 2

PrintN("Ich bin der Long x: "+Str(x))
PrintN("Ich bin der Float y: "+StrF(y))

z.s = "Ich bin der String z"

PrintN(z)

Delay(2000)

Ausgabe:

Ich bin der Long x: 5
Ich bin der Float y: 2.0000000000
Ich bin der String z
```

Man sieht, dass es ganz einfach ist Variablen einen Typ zuzuweisen: Man muss nur hinter den Bezeichner einen Punkt gefolgt von dem ersten Buchstaben des Variablentyps in Kleinschreibung schreiben. Wichtig ist außerdem zu sehen, dass bei der Umwandlung des Floats y in einen String der Befehl *StrF()* benutzt wird. Das hat mit der Formatierung der Ausgabe zu tun, *Str()* würde die Nachkommastellen abschneiden. Für z.B. Quads gibt es ähnliche Befehle. Es wird dann Anstelle des F ein Q geschrieben. Solange also nicht der Standardtyp Long benutzt wird, muss man immer den Variablentyp als großen Buchstaben angegeben.
Die letzte Auffälligkeit ist die Stringvariable. Diese wird über ein kleines s definiert. Ihr kann, wie bei Zahlvariablen auch, ein Wert zugewiesen werden, dieser muss aber natürlich in Anführungstrichen stehen, da es sich um einen String handeln muss. Dementsprechend kann die Variable einfach so an *PrintN()* als Argument übergeben werden, da der Befehl einen String erwartet.

2.3.1 Typumwandlung

An mehreren Stellen wurden schon Typumwandlungen unternommen, z.B. wenn eine Zahlvariable als String ausgegeben wurde. Grundsätzlich ist es möglich jeden Varia-

blenwert in einen anderen Typ umzuwandeln. Dies funktioniert zum einen über die *Str()*-Befehle und zum anderen über die *Val()*-Befehle. Wie bei *Str()* gibt es auch von *Val()* verschiedene Versionen, je nachdem in was für einen Datentyp man den String umwandeln will.

Außerdem kann man in Zahlvariablen ohne explizite Umwandlung in einen anderen Datentyp speichern, PureBasic kümmert sich um alles weitere.

```
; Listing 7: Typumwandlung

OpenConsole()

x.l = 5
y.f = x
PrintN(StrF(y))

y = ValF(Input())
PrintN(StrF(y))

Delay(2000)

Ausgabe:

5.0000000000
2.5
2.5000000000
```

In der sechsten Zeile sieht man, wie ein Longwert einfach in eine Floatvariable gespeichert wird. Andersherum würde es auch funktionieren, die Nachkommastelle würde jedoch abgeschnitten werden. In der neunten Zeile wird der String, der von *Input()* zurückgegeben wird, in einen Float von *ValF()* umgewandelt wird.

Man beachte, dass bei der Eingabe des Floats die amerikanische Konvention gilt, dass also ein Punkt anstatt eines Kommas geschrieben wird.

2.3.2 Strings

Es wurden jetzt schon an mehreren Stellen Strings benutzt. Wie diese grundsätzlich funktionieren, sollte inzwischen klar geworden sein. Es gibt jedoch noch einige andere nützliche Funktionen.

```
; Listing 8: Strings

OpenConsole()

string.s = "Ich bin ein String!"
PrintN(Str(CountString(string, "String")))
```

```
PrintN(LCase(string))

string = InsertString(string, "neuer ",13)
PrintN(string)

string = RTrim(string, "!")
PrintN(string)
PrintN(Left(string,3))
```

Ausgabe:

```
1
ich bin ein string!
Ich bin ein neuer String!
Ich bin ein neuer String
Ich
```

Hier sind einige Befehle zum Arbeiten mit Strings aufgeführt.
CountString() zählt wie oft der angegebene String im Übergebenen vorkommt.
LCase() wandelt alle Großbuchstaben in Kleinbuchstaben um und gibt das Ergebnis zurück.
InsertString() fügt einen neuen String an der angegeben Stelle ein und gibt das Ergebnis ebenfalls zurück.
RTrim() entfernt den angegeben String von Rechts und *Left()* gibt soviele Zeichen vom linken Rand zurück, wie angegeben wurde.

Weitere Befehle für Strings stehen im Referenz-Handbuch.

2.3.3 Aufgaben

1. Es soll ein Programm geschrieben werden, bei dem der Benutzer einen Kreisradius angibt, worauf der Umfang des Kreises angegeben wird. Die Formel für den Umfang lauten $2 \cdot radius \cdot \pi(Pi)$.

2.4 Bedingungen

Bisher waren die kleinen Beispielprogramme sehr undynamisch, soll heißen, sie haben von oben nach unten die Befehle abgearbeitet. Es gibt jedoch sogenannte Kontrollstrukturen, mit denen man den Ablauf dynamischer gestalten kann. In diesem Kapitel wird die If-Else-Klausel vorgestellt.

```
; Listing 9: If-Else-Klausel

OpenConsole()
```

```
x = Val(Input())

If x < 3
  PrintN("x ist kleiner als 3")
ElseIf x = 4
  PrintN("x ist genau 4")
Else
  PrintN("x ist größer als 4")
EndIf

Delay(2000)
```

Ausgabe:

```
2
x ist kleiner als 3

4
x ist genau 4

5
x ist größer als 4
```

Wenn man das Programm kompiliert und ausführt, erwartet es die Eingabe einer ganzen Zahl. Je nachdem, ob man nun einer Zahl kleiner 3, genau 4 oder eine andere eintippt, erhält man eine andere Ausgabe.
Man erhält dieser Verhalten durch sogenannte If-Else-Klauseln, die nach logischen Prinzipien auf die Werte reagieren. Die Schlüsselwörter (Also Wörter, die die Sprache an sich ausmachen und keine Befehle, Konstanten oder Variablen sind, sondern zu ihrer Realisierung vonnöten sind) sind dabei wörtlich aus dem Englischen zu übersetzen: Wenn (*If*) x kleiner als 3 ist..., ansonten, wenn (*ElseIf*) x genau 4 ist..., ansonten (*Else*) in allen anderen Fällen... Das "kleiner alsZeichen (¡) und das ïst gleichZeichen (=) sind sogenannte Vergleichsoperatoren, d.h. sie vergleichen zwei Werte. Werte können hierbei vieles sein: Variablen, Konstanten, Strings, Zahlenwerte. Es gibt natürlich auch ein "größer alsZeichen (¿), ein ïst ungleichZeichen (¡¿) und ein "größer oder gleich bzw. "kleiner oder gleichZeichen (¿= bzw. ¡=).
Wichtig ist zu verstehen, dass die Vergleiche von oben nach unten durchgeführt werden, wenn jedoch ein Vergleich zutrifft (man sagt er ist *wahr*, ansonsten ist er *falsch*), z.B. im ersten Fall x wirklich kleiner als 3 ist, werden die Befehle unter dem Vergleich durchgeführt und die anderen Vergleiche werden nicht mehr durchgeführt, d.h. das Programm springt sofort in der Programmausführung unter *EndIf*.

Es gibt außerdem sogenannte logische Verknüpfungen.

```
; Listing 10: And, Or
```

```
OpenConsole()

x = Val(Input())
y = Val(Input())

If x <= 3 And y >= 2
  PrintN("x ist kleiner oder gleich 3 und y ist größer oder gleich 2")
ElseIf x > 3 Or y < 2
  PrintN("x ist größer als 3 oder y ist kleiner als 2")
EndIf

Delay(2000)
```

Ausgabe:

```
3
2
x ist kleiner oder gleich 3 und y ist größer oder gleich 2

5
2
x ist größer als 3 oder y ist kleiner als 2
```

Das Programm funktioniert wie "Listing 9", jedoch werden diesmal 2 Eingaben erwartet.
Neben den neuen Vergleichsoperatoren, die auch schon oben erwähnt wurden, werden hier die sogenannten logischen Verknüpfungen eingeführt. Auch hier kann wieder wörtlich übersetzt werden. Im ersten Fall muss x kleiner oder gleich 3 sein *und* y muss größer oder gleich 2 sein. Falls der erste Fall jedoch nicht zutrifft, reicht es im zweiten Fall, wenn x größer als 3 ist *oder* y kleiner als zwei ist.
Es gibt außerdem noch die logische Verknüpfung *Not*, die einen Vergleich umkehrt. Eine If-Klausel würde ausgeführt werden, wenn ein Ausdruck falsch wäre. *If Not x ¡ 3* wäre also das gleiche wie *If x ¿= 3*.

2.4.1 Fallunterscheidung mit Select

Eine weitere Möglichkeit ein Programm dynamischer zu gestalten ist die sogenannte Fallunterscheidung. Dafür gibt es das Select-Schlüsselwort.

```
; Listing 11: Select

OpenConsole()

x = Val(Input())
```

```
Select x
  Case 1
    PrintN("x ist gleich 1")
  Case 2, 3, 4
    PrintN("x ist gleich 2, 3 oder 4")
  Case 5 To 10, 11
    PrintN("x hat einen Wert zwischen 5 und 10 oder ist 11")
  Default
    PrintN("x ist größer als 11 oder kleiner als 1")
EndSelect

Delay(2000)

Ausgabe:

1
x ist gleich 1

4
x ist gleich 2, 3 oder 4

-1
x ist größer als 11 oder kleiner als 1

9
x hat einen Wert zwischen 5 und 10 oder ist 11
```

Wieder wird die Eingabe von x erwartet. Dann wird der Wert von x unterscheidet. Es werden immer die Befehle ausgeführt, die unter dem eingetroffenen Fall stehen. Wenn x also 1 ist, wird der Befehl *PrintN("x ist gleich 1")* ausgeführt und danach springt das Programm direkt zu *Delay(2000)*.
Außerdem ist es möglich mehrere Fallunterscheidungen zu kombinieren, damit nicht jeder Fall einzeln betrachtet werden muss, wenn die gleichen Befehle ausgeführt werden sollen. Dafür schreibt man entweder die Fälle mit Kommata getrennt oder benutzt das *To*-Schlüsselwort, der nur für Zahlen funktioniert und alle Zahlen von ... bis (*To*) ... unterscheidet.
Wenn kein Fall zutrifft, wird der optionale Standardfall (*Default*) ausgeführt.
Zuletzt sei gesagt, dass neben Strings nur ganze Zahlen unterschieden werden. Wenn man einen Float übergibt, wird die Zahl zur nächsten Ganzzahl abgerundet.

2.4.2 Aufgaben

1. Es soll das Umfangsprogamm aus dem vorigen Kapitel so abgeändert werden, dass es einen Umfang von 0 ausgibt, wenn ein negativer Radius eingegeben wird.

2. Es soll ein Programm geschrieben werden, bei dem man die Geschwindigkeit eines Autos eingibt und das darauf ausgibt, ob der Wagen zu schnell fährt und wenn, wieviel zu schnell. Als Tempolimit kann 50 km/h angenommen werden.

2.5 Schleifen

Es gibt Fälle, in denen man einen Programmteil mehrmals hintereinander wiederholen will. Dafür gibt es sogenannte Schleifen.

2.5.1 For-Schleife

Die For-Schleife führt einen Block von Befehlen eine festgelegte Anzahl durch.

```
; Listing 12: For-Schleife

OpenConsole()

For x = 1 To 10
  PrintN(Str(x))
Next

Delay(2000)
```

Ausgabe:

```
1
2
3
4
5
6
7
8
9
10
```

Die Ausgabe erzeugt die Zahlen 1 bis 10 untereinander geschrieben. Man übergibt *For* eine Variable, die direkt hinter *For* einen Wert zugewiesen bekommen muss. Dann schreibt man bis zu welchem Wert die Variable hochgezählt werden soll. Es wird jeder

Befehl bis *Next* ausgeführt, dann wird zu x eine 1 hinzuaddiert und der Block an Befehlen wird erneut ausgeführt. Eine Ausführung des gesamten Blocks ist eine sogenannte *Iteration*. Sobald x größer als 10 ist, geht die Ausführung unter *Next* weiter.

Es ist auch möglich, die Zahl, die mit jeder Iteration zu x hinzuaddiert wird, zu modifizieren.

```
; Listing 13: Step

OpenConsole()

For x = 1 To 10 Step 2
  PrintN(Str(x))
Next

Delay(2000)
```

Ausgabe:

```
1
3
5
7
9
```

Über das *Step*-Schlüsselwort wird festgelegt, dass in diesem Fall immer 2 anstatt 1 hinzuaddiert wird. *Step* kann auch negativ sein.

```
; Listing 14: Negativer Step

OpenConsole()

For x = 10 To 1 Step -1
  PrintN(Str(x))
Next

Delay(2000)
```

Ausgabe:

```
10
9
8
7
6
5
4
```

```
3
2
1
```

2.5.2 While und Repeat

Neben der For-Schleife gibt es außerdem die While- und die Repeat-Schleife.

```
; Listing 15: While

OpenConsole()

While x <= 10
  PrintN(Str(x))
  x+1
Wend

Delay(2000)
```

Ausgabe:

```
0
1
2
3
4
5
6
7
8
9
10
```

Der While-Schleife muss ein Vergleich übergeben werden und die Schleife wird solange ausgeführt, wie dieser zutrifft. D.h. es gibt auch Fälle, in denen die Schleife garnicht ausgeführt wird, in Gegensatz zur For-Schleife, nämlich wenn x schon vor der Ausführung der Schleife größer als 10 wäre. Außerdem sieht man an diesem Beispiel, dass eine Variable ohne Initialisierung den Wert 0 hat.

```
; Listing 16: Repeat

OpenConsole()

Repeat
  PrintN(Str(x))
```

```
  x+1
Until x > 10

Delay(2000)
```

Ausgabe:

```
0
1
2
3
4
5
6
7
8
9
```

Die Repeat-Schleife unterscheidet sich insofern von der While-Schleife, als dass die Schleife immer mindestens einmal ausgeführt wird und außerdem solange ausgeführt wird *bis* ein bestimmer Fall eintritt und nicht *solange* einer zutrifft.

Es ist möglich jede Schleife als eine While-Schleife zu schreiben.

2.5.3 Break und Continue

Bisher haben die Schleifen einfach strikt den Befehlsblock abgearbeitet. Es ist aber möglich, diese Ausführung dynamischer zu gestalten.

```
; Listing 17: Break und Continue

OpenConsole()

For x = 1 To 10
  If x = 5
    Continue
  ElseIf x = 8
    Break
  EndIf

  PrintN(Str(x))
Next

Delay(2000)
```

Ausgabe:

```
1
2
3
4
6
7
```

Man sieht das nur die Zahlen von 1 bis 7 ausgegeben werden, ohne die 5 und auch ohne die Zahlen 8 bis 10, obwohl man es von der For-Anweisung erwarten würde. Dies liegt an den Schlüsselwörtern *Continue* und *Break*. *Continue* springt sofort zur nächsten Iteration, ohne dass der restiliche Befehlsblock ausgeführt wird. *Break* bricht die gesammte Schleife sofort ab, ohne das die nachfolgenden Iterationen noch beachtet werden.

2.5.4 Aufgaben

1. Es soll ein Programm geschrieben werden, dass nach einem String und einer ganzen positiven Zahl n fragt und den String n-mal ausgibt.

2.6 Arrays, Listen und Maps

Es gibt Fälle, in denen man mehrere zusammenhängende Werte speichern möchte. Nun könnte man für jeden Wert eine eigene Variable anlegen, was jedoch unvorteilhaft wäre und zu Spaghetti-Code führen kann, also zu sehr schwer leslichen und unübersichtlichen Code. Dafür gibt es Arrays.

```
; Listing 18: Arrays

OpenConsole()

Dim array.l(2)

For x = 0 To 2
  array(x) = x
Next

For x = 0 To 2
  PrintN(Str(array(x)))
Next

Delay(2000)
```

Ausgabe:

```
0
1
2
```

In der 5. Zeile wird das Array deklariert und definiert. Dazu gibt es das Schlüsselwort *Dim*. Man schreibt dahinter den Bezeichner für das Array und in runden Klammern, die zum Bezeichner gehören, den höchsten sogenannten *Index*. Der Index wird benötigt, um auf die einzelnen Variablen, die im Array gespeichert sind, zuzugreifen. Der niedrigste Index ist 0. Die Typzuweisung soll verdeutlichen, wie man Arrays unterschiedlichen Typs erzeugt, nämlich genauso, wie bei normalen Variablen auch. Ein Array nimmt immer die Anzahl der Elemente mal die Größe einer einzelnen Variable des Arrays im Speicher ein, in diesem Fall also 12 Byte, da es sich um ein Array von Longs handelt (3 x 4 Byte).
In der ersten For-Schleife sieht man, wie ein Zugriff auf ein Array aussieht: Man schreibt in die Klammern des Bezeichners den Index auf den man zugreifen will. Durch die For-Schleife greift man nacheinander auf alle Indizes zu. Genauso kann man auch die Werte der Variablen im Array abrufen, was in der zweiten For-Schleife geschieht.

2.6.1 Größenänderung

Es ist möglich die Größe eines Arrays im Programm zu ändern.

```
; Listing 19: ReDim

OpenConsole()

Dim a.l(2)

For x = 0 To 2
  array(x) = x
Next

For x = 0 To 2
  PrintN(Str(a(x)))
Next

ReDim a(3)

a(3) = 3
PrintN(Str(a(3)))

Delay(2000)

Ausgabe:

0
```

```
1
2
3
```

Das Beispiel unterscheidet sich insofern von dem ersten, als dass weiter unten das Array um ein Element erweitert wird, nämlich über das Schlüsselwort *ReDim*. Man schreibt dabei in die Klammern des Bezeichners die neue Größe des Arrays. Wenn die neue Größe kleiner ausfällt als die vorige wird der Rest äbgeschnitten".

2.6.2 Mehrdimensionale Arrays

Mehrdimensionale Arrays kann man sich als Arrays von Arrays vorstellen. Unter jedem Index des ersten Arrays ist ein weiteres Array. Bildlich kann man sich das so vorstellen: Ein einfaches Array ist eine Reiche von Variablen, ein zweidimensionales Array ist dann ein Schachbrett-ähnliches Gebilde und ein dreidimensionales ein Raum. Ab vier Dimensionen hinkt der Vergleich jedoch, dann sollte man sich der Baumdarstellung als Array von Arrays von Arrays von Arrays... bemühen.

```
; Listing 20: Arrays

OpenConsole()

Dim a.l(2,2)

For x = 0 To 2
  For y = 0 To 2
    a(x,y) = x
  Next
Next

For x = 0 To 2
  For y = 0 To 2
    Print(Str(a(x,y)))
  Next
PrintN("")
Next

Delay(2000)

Ausgabe:

000
111
222
```

Ein mehrdimensionales Array wird erzeugt, indem man in die Klammern mehrere maximale Indizes getrennt durch Kommata schreibt. In diesem Fall wurde z.B. ein zweidimensionales Array erzeugt. Wenn man ein dreidimensionales erzeugen möchte, müsste ein weiterer maximaler Index, wieder durch ein Komma getrennt, hinzugefügt werden. Außerdem wurde hier die sogenannte *Verschachtelung* angewandt. Es wurde eine For-Schleife in eine For-Schleife geschrieben, d.h. Es wird in der ersten Iteration der äußeren Schleife die komplette innere ausgewertet. In der zweiten wird wieder die komplette innere ausgewertet usw. Dies macht man sich zunutze, um das mehrdimensionale Array zu füllen: Der erste Index bleibt gleich und es wird das komplette darunterliegende Array gefüllt. Dann wird das nächste gefüllt usw.

Man beachte, dass bei mehrdimensionalen Arrays nur die Größe der letzten Dimension geändert werden kann. Im Fall aus *Listing 20* müsste man schreiben *ReDim a(2,neue_größe)*.

2.6.3 Linked Lists

Linked Lists (Listen) sind dynamische Datenstrukturen, die sich dadurch abheben, dass dynamisch Elemente hinzugefügt und entfernt werden können. Im Prinzip, sind es also dynamische Arrays, auch wenn die interne Darstellung eine ganz andere ist.

```
; Listing 21: Linked Lists

OpenConsole()

NewList l.l()

For x = 1 To 10
  AddElement(l())
  l() = 1
Next

SelectElement(l(),2)
InsertElement(l())

l() = 15

FirstElement(l())
For x = 1 To 11
  PrintN(Str(l()))
  NextElement(l())
Next

Delay(2000)

Ausgabe:
```

```
1
1
15
1
1
1
1
1
1
1
1
```

Es werden untereinander 11 Zahlen ausgegeben, nämlich zehn Einsen, sowie an der dritten Stelle eine 15.
Zuerst wird eine neue Liste mittels *NewList* erzeugt, wobei die Klammern wie bei Arrays zum Bezeichner gehören. Mit *AddElement()* wird immer wieder ein neues Element zur Liste hinzugefügt, das danach einen Wert zugewiesen bekommt. Man sieht das kein Index von Nöten ist, vielmehr verwaltet PureBasic für jede Liste einen internen Index und man muss selber mit den verfügbaren Befehlen dafür sorgen, dass auf das richtige Element zugegriffen wird. *AddElement()* setzt immer den Index auf das neue Element.
Mit *SelectElement()* kann man den Index der Liste ändern, wobei wie bei Arrays das erste Element den Index 0 hat. InsertElement() fügt ein neues Element vor dem aktuellen ein und setzt den Index auf dieses.
FirstElement() setzt den Index zurück auf das erste Element, sodass anschließend mit der For-Schleife die Liste ausgegeben werden kann, wobei *NextElement()* den Index auf das nächste Element setzt.

Weitere Befehle stehen in der Handbuch-Referenz.

2.6.4 Maps

Maps sind Datenstrukturen, in denen die Elemente nicht über Indizes referenziert werden, sondern über Schlüssel.

```
; Listing 22: Maps

OpenConsole()

NewMap initialien.s()

initialien("HP") = "Hans Peter"
initialien("KS") = "Kirill Schuschkow"

PrintN(initialien("HP"))
PrintN(initialien("KS"))
```

```
Delay(2000)
```

Ausgabe:

```
Hans Peter
Kirill Schuschkow
```

Maps werden ähnlich wie Listen angelegt, durch das Schlüsselwort *NewMaps*. Jedoch muss man neue Element nicht über einen Add-Befehl hinzufügen, sondern es reicht auf die Map mit dem neuen Schlüssel zuugreifen und gleichzeitig mit dem Wert für diesen Schlüssel zu initialisieren; das Element wird dann mit dem Wert neu angelegt. Über diese Schlüssel kann dann auf die Elemente zugegriffen werden. Die PureBasic-Referenz stellt vielseitige Befehle zu Maps vor.

2.6.5 ForEach-Schleife

Mit der ForEach-Schleife ist es möglich alle Elemente von Listen und Maps einfach zu durchlaufen.

```
; Listing 23: ForEach-Schleife

OpenConsole()

NewMap initialien.s()

initialien("HP") = "Hans Peter"
initialien("KS") = "Kirill Schuschkow"

ForEach initialien()
  PrintN(initialien())
Next

Delay(2000)
```

Ausgabe:

```
Hans Peter
Kirill Schuschkow
```

Die ForEach-Schleife duchläuft alle Elemente einer Maps, ohne dass man die Schlüssel eingeben muss. *initialien()* repräsentiert dann immer das aktuelle Element, das dementsprechend über diesen Bezeichner verändert werden kann.

Die ForEach-Schleife funktioniert auf die gleiche Art und Weise mit Linked Lists.

2.6.6 Aufgaben

1. Es soll ein Programm geschrieben werden, dass ein Menü anzeigt. Es soll die Möglichkeit geben ein zweidimensionales Feld auszugeben, das standardmäßig mit Nullen gefüllt ist, oder eine Koordinate einzugeben. Wenn eine Koordinate eingegeben wird, soll auf das Feldelement, das durch die Koordinate repräsentiert wird, eine 1 addiert werden.

3 Fortgeschrittene Themen

3.1 Strukturen

Strukturen sind dazu da, um Variablen sinnvoll zu ordnen. Man kann mit ihnen eigene Datentypen definieren und so alle Daten eines einzelnen Bestandes zusammenfassen.

```
; Listing 24: Strukturen, With

Structure Person
  name.s
  alter.l
EndStructure

OpenConsole()

Dim alle.Person(2)

For x = 0 To 2
  With alle(x)
    \alter = x
    \name = "Hans"
  EndWith
Next

For x = 0 To 2
  PrintN("Alter: "+Str(alle(x)\alter))
  PrintN("Name: "+alle(x)\name)
Next

Delay(2000)

Ausgabe:

Alter: 0
Name: Hans
Alter: 1
Name: Hans
Alter: 2
```

```
Name: Hans
```

Zu Anfang wird der Strukturtyp im *Structure : EndStructure*-Block definiert. Der neu erzeugte Typ erhält den Namen *Person* und enthält die Stringvariable "nameünd die Longvariable älter"(Strukturvariablen). Dann wird ein Array von diesem neuen Typ erzeugt und gefüllt. Dazu ist grundsätzlich nicht das Schlüsselwort *With* vonnöten. *With* wird eine definierte Struktur übergeben und in diesem Block wird über einen Backslash und den Strukturvariablennamen auf die Strukturvariable zugegriffen. Wie man in der zweiten For-Schleife sieht, kann auch ohne With auf eine Strukturvariable zugegriffen werden, indem man vor den Backslash noch die definierte Struktur schreibt.

3.2 Prozeduren

Prozeduren sind ein Mittel, um immer wieder benötigten Code einfach zur Verfügung zustellen. Anstatt den gleichen Code immer und immer wieder in die Quelldatei zu schreiben, definiert man eine Prozedur. Prozeduren sind im Grunde nichts anderes, als die schon vorhandenen Befehle von PureBasic, *PrintN()* ist z.B. eine.

```
; Listing 25: Prozeduren

Procedure.l square(x.l)
  ProcedureReturn x*x
EndProcedure

OpenConsole()

PrintN(Str(square(2)))

Delay(2000)

Ausgabe:

4
```

Dieses einfache Beispiel verdeutlicht, wozu Prozeduren in der Lage sind: Anstatt immer wieder *x*x* einzutippen, schreibt man *square(x)*, wodurch der Quellcode insgesamt lesbarer wird, da immer sofort erkennbar ist, was gemeint wurde. Natürlich kann man auch weitaus komplexeren Code in eine Prozedur schreiben.
Hinter dem Schlüsselwort *Procedure* wurde ein Typ übergeben (Long). Dieser ist wichtig, da er definiert, was für einen Typ der Rückgabewert hat. Der Rückgabewert ist das, was hinter *ProcedureReturn* steht. Nach der Auswertung der Prozedur erhält *Str()* diesen Wert und kann ihn weiterverarbeiten, genauso wie *Str()* einen String als Rückgabewert hat, der von *PrintN()* weiter verarbeitet werden kann. Nachdem *ProcedureReturn* aufgerufen wurde, springt die Programmausführung aus der Prozedur wieder zu der Stelle, an der die Prozedur aufgerufen wurde.

In den Klammern des Prozedurenbezeichners ist ein sogenanntes Argument. Dieses erhält einen Namen und einen Typ. Beim Aufruf der Prozedur muss immer ein Long als Argument übergeben werden, genauso wie man *Str()* einen Long übergeben muss, damit dieser in einen String umgewandelt werden kann. Eine Prozedur kann auch mehrere Argumente erhalten. Es ist zu beachten, dass nicht die Variable an sich übergeben wird, sondern der Wert der Variable in eine interne Prozedurvariable mit dem Argumentennamen als Bezeichner kopiert wird.
Der Rückgabetyp, der Bezeichner und die Argumente ergeben zusammen den sogenannten *Prozedurkopf*.

3.2.1 Sichtbarkeit

Sichtbarkeit bedeutet, dass eine Variable, die außerhalb einer Prozedur definiert wurde, unter Umständen nicht in der Prozedur verfügbar ist.

```
; Listing 26: Sichtbarkeit

Procedure scope()
  x = 5
  PrintN(Str(x))
EndProcedure

OpenConsole()

x = 2
scope()
PrintN(Str(x))

Delay(2000)
```

Ausgabe:

```
5
2
```

Erst wird 5 ausgegeben und dann die 2, obwohl in der Prozedur $x = 5$ steht. Man könnte also annehmen, dass zweimal eine 5 augegeben werden müsste. Dies liegt daran, dass x in der Prozedur ein anderes ist, als das außerhalb. Es gibt Schlüsselwörter, wie *Global*, *Protected* und *Shared*, die dazu da sind, diesen Umstand zu umgehen, die Benutzung dieser ist jedoch in der modernen Programmierung verpönnt. Man arbeitet heutzutage über Zeiger, auf die später eingegangen wird.

3.2.2 Static

Das Schlüsselwort *Static* wird dazu benutzt, um eine Variable in einer Funktion einmal zu definieren und zu initialiseren, jedoch kein weiteres mal.

```
; Listing 27: Static

Procedure static_beispiel()
  Static  a = 1
  a+1
  Debug a
EndProcedure

OpenConsole()

static_beispiel()
static_beispiel()
static_beispiel()

Delay(2000)
```

Ausgabe:

```
2
3
4
```

Die Ausgabe ergibt hintereinander 2, 3 und 4, obwohl in der Prozedur *a = 1* steht. Dies liegt am Schlüsselwort *Static*, dass eine neue Definition und Initialisierung verhindert.

3.2.3 Arrays als Argument

Es ist auch möglich Arrays als Argument zu übergeben.

```
 ; Listing 28: Array als Argument

Procedure array_argument(Array a(1))
  For x = 2 To 0 Step -1
    a(x) = 2-x
  Next
EndProcedure

OpenConsole()

Dim a(2)
For x = 0 To 2
```

```
  a(x) = x
Next

array_argument(a())

For x = 0 To 2
  PrintN(Str(a(x)))
  Debug a(x)
Next

Delay(2000)
```

Ausgabe:

```
2
1
0
```

Wenn man ein Array als Argument übergibt, muss man in den Klammern des Arraybezeichners die Anzahl der Arraydimensionen angeben und vor den Bezeichner das Schlüsselwort *Array* schreiben. Wichtig ist, dass Arrays nicht wie normale Variablen kopiert werden, sondern als sogenannte Referenz, d.h. das Array in der Prozedur ist *dasselbe* Array wie außerhalb der Prozedur. Alle Änderungen die in der Prozedur am Array vorgenommen werden, sind auch außerhalb der Prozedur sichtbar, deswegen werden die Zahlen rückwärts ausgegen, weil sie in der Prozedur in das gleich Array geschrieben werden. Deshalb muss man außerdem Arrays nicht als Rückgabewert zurückgeben, genauer: es geht garnicht.

Die gleichen Regeln gelten auch für Maps und Listen.

3.2.4 Declare

Es gibt Fälle, in denen man eine Prozedur aufrufen möchte, diese jedoch noch nicht definiert ist. Hier kommen Prozedurprototypen ins Spiel.

```
; Listing 29: Declare

OpenConsole()

Declare.l square(x.l)

PrintN(Str(square(2)))

Procedure.l square(x.l)
  ProcedureReturn x*x
EndProcedure
```

```
Delay(2000)
```

Ausgabe:

4

In diesem Fall wird die Prozedur *square()* aufgerufen, bevor sie definiert wurde. Deshalb deklariert man sie vorher mit dem Schlüsselwort *Declare*. wodurch erst dieses Verhalten möglich wird. Der Prozedurkopf muss dabei genau der gleiche sein, wie der bei der Definition der Prozedur.
Dieses Verhalten ist vorallem dann hilfreich, wenn man eine Prozedur in einer anderen aufrufen möchte, obwohl sie erst unter dieser definiert wird.

3.2.5 Aufgaben

1. Es soll ein Programm geschrieben werden, in dem einer Prozedur ein Array von Zahlen übergeben wird und die alle Zahlen ausgibt, die größer als der Durchschnitt sind. Der Durchschnitt berechnet sich über die Gesamtsumme der Zahlen geteilt durch die Anzahl der Zahlen. Der Inhalt des Arrays kann fest einprogrammiert werden und muss nicht unbedingt vom Benutzer eingegeben werden.

3.3 Code-Auslagerung

Sobald Projekte eine gewisse Größe erreichen, wird es sinnvoll den Code auf mehrere Dateien aufzuteilen.

ausgelagert.pb

```
Procedure ausgelagert()
  PrintN("Ich bin eine ausgelagerte Prozedur")
EndProcedure
```

main.pb

```
; Listing 30: Code-Auslagerung

IncludeFile "./ausgelagert.pb"

OpenConsole()

ausgelagert()

Delay(2000)
```

Ausgabe:

```
Ich bin eine ausgelagerte Prozedur
```

Nachdem man den Code in die zwei Dateien eingetippt und im gleichen Ordner abgespeichert hat, kann die Prozedur *ausgelagert()* in der Hauptdatei aufgerufen werden. Dies liegt am Schlüsselwort *IncludeFile*, dass den Inhalt der übergebenen Datei an genau dieser Stelle einfügt. Theoretisch kann man auch normale Befehle, Variablen, Arrays etc. auf diese Weise auslagern, normalerweise lagert man aber Prozeduren aus, die aus der Hauptdatei aufgerufen werden sollen.

Mit dem Schlüsselwort *XIncludeFile* wird das Gleiche erreicht, jedoch verhindert dieses im Gegensatz zum anderen, dass eine Datei mehrmals eingefügt wird.

3.4 Zeiger

Viele Programmier bezeichnen Zeiger als eines der am schwierigsten zu verstehenden Themen im Bereich Programmierung. Dabei sind Zeiger ganz einfach zu verstehen, wenn man sich klar macht, wie Variablen und andere Daten im Arbeitsspeicher liegen.

Jede Variable im Arbeitsspeicher hat eine *Adresse*. Wenn man sich den Arbeitspeicher bildlich wie eine Reihe von verschieden großen Kisten (je nachdem welchen Typ die Variable hat) vorstellt, dann liegen die Variablen in diesen, wobei jeder dieser Kisten eine Adresse hat, wie eine Hausnummer. Der Computer arbeitet intern auf der Maschinensprachenebene mit diesen Adressen, um auf die Variablen zuzugeifen, er benutzt dafür nicht die Bezeichner. Man benutzt nun Zeiger um dies ebenfalls wie der Computer zu tun. Im Zeiger speichert man also die Adresse einer Variable, um über jenen auf diese zuzugreifen. Man nennt diesen Zugriff dann *Dereferenzierung*, denn der Zeiger stellt eine Referenz zur Variable dar.

```
; Listing 30: Zeiger

Procedure change(*ptr2.l)
  PokeL(*ptr2, 6)
EndProcedure

x = 5

PrintN(Str(x))

*ptr = @x
change(*ptr)
PrintN(Str(PeekL(*ptr)))

Ausgabe:

5
6
```

Man definiert eine Funktion, die einen Zeiger auf eine Longvariable (.l) als Argument hat, d.h. sie erwartet die Adresse einer Longvariable. Man definiert die Variable x und initialisiert sie mit 5. Nachdem man sie ausgegeben hat, definiert man den Zeiger **ptr* und initialisiert ihn mit der Adresse von x. Man erhält die Adresse über den Operator @. Das Sternchen gehört fest zum Bezeichner des Zeigers. In der Prozedur wird der Zeiger über *PokeL()* dereferenziert, sodass auf die eigentliche Variable x zugegriffen werden kann, mit dem Ziel in diese einen neuen Wert zu schreiben. Wenn man auf Variablen anderen Typs dereferenzieren will, muss man dementsprechen das L austauschen, genauso wie bei *Str()* und *Val()*. Man muss *PokeL()* dabei eine Adresse und einen neuen Wert übergeben. Mit *PeekL()* kann man ebenfalls einen Zeiger derefenzieren, erhält jedoch den Wert der Variable als Rückgabewert. Für das L von *PeekL()* gilt das Gleiche wie bei *PokeL()*.

3.5 GUI-Programmierung

Dieses Kapitel soll eine Einführung in die GUI-Programmierung mit PureBasic bieten. Dabei sollen nicht alle Befehle behandelt werden, wenn man jedoch die Grundlagen verstanden hat, kann man sich die restlichen Befehle je nach Bedarf über die Referenz aneignen.

```
; Listing 32: Ein Fenster

OpenWindow(0, 0, 0, 200, 200, "Ein Fenster",\\
 #PB_Window_ScreenCentered|#PB_Window_SystemMenu)
TextGadget(0, 0, 0, 100, 20, "TextGadget")

Repeat : Until WindowEvent() = #PB_Event_CloseWindow
```

Hinweis: Die zwei Backslashes bedeutet, dass die nächste Zeile noch in die obere gehört!

Nach der Kompilierung wird ein simples Fenster mit dem Text *TextGadget* dargestellt, dass nichts anderes tut, außer über "x"geschlossen zu werden.
Über *OpenWindow()* wird das Fenster geöffnet, wobei die Argumente darstellen, wie das Fenster dargestellt werden soll. Von Links, nach Rechts: Fensternummer, Position des Fensters in x und in y Richtung, Breite und Höhe, sowie der Titel. Die beiden Konstanten sind sogenannte *Flags*, die über ein binäres Öder"verknüpft werden. Die erste bedeutet, dass das Fenster in der Bildschirmmitte angezeigt wird und die zweite sorgt dafür, dass das Schließensymbol angezeigt wird.
In der nächsten Zeile ist ein sogeanntes Textgadget. Alles in Fenstern wird über Gadgets dargestellt. Die Argumente sind die gleichen, wie beim Fenster. Weitere Flags für Fenster und die jeweiligen Gadgets stehen in der Referenz.
Die letzte Zeile ist eine Repeat-Schleife, wobei die Zeilenorientierung des Compilers über den Doppelpunkt umgangen wird. Durch ihn wird es möglich, mehrere Befehl in eine Zeile zu schreiben, was insbesondere bei solch kurzen Befehlen Sinn macht. Der

Ausdruck hinter *Until* sorgt dafür, dass das Fenster erst geschlossen wird, genauer gesagt, dass das Programm erst beendet wird, denn darunter sind keine Befehle mehr, wenn das "x"geklickt wird. Dieses erzeugt nämlich ein sogenanntes Event, dass eine dynamische Programmsteuerung möglich macht. *WindowEvent()* gibt einen Wert zurück, wenn ein Event ausgelöst, der durch die Konstante repräsentiert wird. In diesem Fall wurde also beim Klicken auf das "xëin Event ausgelöst, das durch die Konstante *#PB_Event_CloseWindow* repräsentiert wird. Der Ausdruck wird wahr und die Schleife beendet.

3.5.1 Buttons

Buttons werden ebenfalls über Gadgets repräsentiert.

```
; Listing 33: Button

OpenWindow(0, 0, 0, 200, 200, "Ein Fenster", #PB_Window_\\
ScreenCentered|#PB_Window_SystemMenu)
TextGadget(0, 0, 0, 200, 20, "TextGadget")
ButtonGadget(1, 0, 30, 100, 30, "Klick mich!")

Repeat
  Select WindowEvent()
    Case #PB_Event_CloseWindow
      End
    Case #PB_Event_Gadget
      If EventGadget() = 1
        SetGadgetText(0, "Button geklickt")
      EndIf
  EndSelect
ForEver
```

Es wird genauso wie in *Listing 32* ein Fenster geöffnet, wobei noch ein Button hinzugefügt wird. Die Gadgetnummer muss dabei natürlich eine andere, als die vom Textgadget sein. Wenn man nun auf den Button klickt, ändert sich der Inhalt des Textgadgets. In der Repeat-Schleife, wird wieder auf Events reagiert. Neu ist zuerst das Schlüsselwort *ForEver*, dass dafür sorgt, dass die Repeat-Schleife nie abbricht, außer durch Break oder wie in diesem Fall durch End.
Man mach sich hier die Fallunterscheidung zunutze: *WindowEvent()* gibt wieder einen Wert zurück, wenn ein Wert ausgelöst wird, der durch *Select* unterschieden wird. Wenn das Fenster geschlossen wurde, wird das Programm durch das Schlüsselwort *End* beendet. Wen jedoch ein Event von einem Gadget ausgelöst wurde, wird der Wert *#PB_Event_Gadget* zurückgegeben. Mit dem Befehl *EventGadget()* wird die Gadgetnummer des auslösenden Gadgets zurückgegeben. Wenn es der Button war, wird über den Befehl *SetGadgetText()* der Text des Textgadgets geändert, wobei die Argumente die Gadgetnummer und der neue Text sind.

3.5.2 Stringgadgets

Bisher passierte noch nicht viel. Interessant wird es, sobald Stringgadgets ins Spiel kommen, mit denen man Benutzereingaben empfangen kann.

```
; Listing 34: Stringgadgets

OpenWindow(0,200,200,200,150,"Idealgewicht",#PB_Window_\\
SystemMenu|#PB_Window_ScreenCentered)
StringGadget(0,0,0,200,25,"Körpergröße eingeben in cm")
StringGadget(1,0,35,200,25,"Geschlecht (m/w)")
ButtonGadget(2,0,75,200,30,"Idealgewicht ausrechnen")
StringGadget(3,0,115,200,25,"Idealgewicht", #PB_String_\\
ReadOnly)

Repeat
  Select WindowEvent()
    Case #PB_Event_CloseWindow
      End
    Case #PB_Event_Gadget
      If EventGadget() = 2
        If GetGadgetText(1) = "m"

          SetGadgetText(3, StrF((ValF(GetGadgetText(0))-\\
          100) * 0.93) + " bis " +StrF((ValF(GetGadgetText(0)\\
          )-100)*0.97) + " kg")

        ElseIf GetGadgetText(1) = "w"

          SetGadgetText(3, StrF((ValF(GetGadgetText(0))-100)\\
           * 0.88) + " bis " +StrF((ValF(GetGadgetText(0))-100)\\
           *0.92) + " kg")

        Else
         MessageRequester("Fehler","Bitte echtes Geschlecht\\
          eingeben")
       EndIf
     EndIf
  EndSelect
ForEver
```

Abermals wird ein Fenster geöffnet. Neu sind die Eingabefelder, die durch Stringgadgets repräsentiert werden. In diese kann man Strings eingeben, die mit *GetGadgetText()* abgerufen werden, wobei das Argument die Gadgetnummer ist. *GetGadgetText()* ist also das Gegenstück zu *SetGadgetText()*.

Das Programm tut nichts anderes, als das Idealgewicht anhand von Größe und Geschlecht auszurechnen. Sobald der Button gedrückt wird (*EventGadget() = 2*), wird mit den genannten Befehlen abgerufen, welches Geschlecht eingegeben wurde und dann mit der Körpergröße das Idealgewicht ausgerechnet, das mit *SetGadgetText()* im unteren Stringgadget angezeigt wird.
Die Konstante beim unteren Stringgadget verhindert, dass in dieses Dinge eingegeben werden können.
Der letzte neue Befehl ist *MessageRequester()*. Dieser tut nichts anderes, als ein Fenster mit einem Titel und einem Text zu öffnen, das über einen Button geschlossen wird. Das erste Argument ist der Titel, das zweite der Text.

Mit diesen wenigen Beispielen hat man schon einen guten Überlick über die GUI-Programmierung gewonnen, sofern man sie durchgearbeitet und verstanden hat. Die wenigen anderen Elemente sind in der PureBasic-Referenz erläutert und können sich bei Bedarf angeeignet werden.

3.6 Zeichnen

Es ist möglich Formen und Linien zu zeichnen. Dies ist z.B. nützlich, wenn man ein Simulationsprogramm programmerien will oder einen Funktionsplotter. Plakativ soll hier zweidimensional gezeichnet werden, in der PureBasic-Vollversion ist es auch möglich 3D-Objekte zu erstellen.

```
; Listing 35: Zeichnen

OpenWindow(0,0,0,400,400,"Umkehrungen",#PB_Window_SystemMenu\\
|#PB_Window_ScreenCentered)

Procedure.f turn_x(x.f, y.f, degree.f)
  ProcedureReturn Cos(Radian(degree))*x-Sin(Radian(degree))*y
EndProcedure

Procedure.f turn_y(x.f,y.f,degree.f)
  ProcedureReturn Sin(Radian(degree))*x+Cos(Radian(degree))*y
EndProcedure

StartDrawing(WindowOutput(0))
x1.f=100
y1.f=100
x2.f=140
y2.f=140
x3.f=100
y3.f=140
degree.f = 1
```

```
Repeat
  LineXY(x1+200,y1+200,x2+200,y2+200,RGB(0,0,0))
  LineXY(x1+200,y1+200,x3+200,y3+200,RGB(0,0,0))
  LineXY(x2+200,y2+200,x3+200,y3+200,RGB(0,0,0))
  LineXY(x1+200,y1+200,200,200,RGB(0,0,0))
  Circle(200,200,2,RGB(0,0,0))

  x1_new.f=turn_x(x1,y1,degree)
  y1_new.f=turn_y(x1,y1,degree)
  x2_new.f=turn_x(x2,y2,degree)
  y2_new.f=turn_y(x2,y2,degree)
  x3_new.f=turn_x(x3,y3,degree)
  y3_new.f=turn_y(x3,y3,degree)

  x1=x1_new
  y1=y1_new
  x2=x2_new
  y2=y2_new
  x3=x3_new
  y3=y3_new

  Delay(10)
  Box(0,0,400,400)
Until #PB_Event_CloseWindow = WindowEvent()
```

Dieses Programm öffnet ein Fenster, in dem in der Mitte ein Kreis gezeichnet wird (*Circle()*), von dem eine Linie ausgeht, an der ein Dreieck hängt (*LineXY()*). Die Linie und das Dreieck drehen sich nun im Uhrzeigersinn.
Mit *StartDrawing()* wird ausgesagt, dass nun begonnen werden soll etwas zu zeichnen. Das Argument ist dabei die Form der Ausgabe, in diesem Fall ein Fenster. *WindowOutput()* muss wiederum als Argument die Fensternummer übergeben werden, die zur Zeichenausgabe dienen soll. Für *StartDrawing()* gibt es noch weitere Argumente, um z.B. eine Ausgabe zu drucken. Wenn fertig gezeichnet wurde und man noch andere Dinge im Programm ausführen will, sollte man *StopDrawing()* ausführen, da damit nicht mehr benötigte Ressources freigegeben werden.
Mit *Circle()* wird wie gesagt ein Kreis gezeichnet, wobei die ersten beiden Argumente die x- bzw. die y-Position darstellen und die anderen Argumente den Radius und die Farbe in RGB-Schreibweise. Letzteres heißt, dass die erste Zahl den Rot-, die zweite den Grün- und die letzte den Blauanteil darstellt. Bei der x-y-Position wird 200 hinzuaddiert, weil die Prozeduren zum Drehen des Zeigers, diesen um die linke obere Fensterecke drehen.
Mit *LineXY()* werden die Linien gezeichnet, wobei man jeweils zwei Punkte, die verbunden werden sollen, mit x-y-Koordinate angeben muss. Zusätzlich muss wieder die Farbe angegeben werden.
Darunter werden die Punkte, die verbunden werden sollen gedreht. Die Prozeduren ba-

sieren hierbei auf Vektorrechnung. Die neuen Werte müssen zuerst in Zwischenvariablen gespeichert werden, da sowohl für die neue x-, als auch für die neue y-Koordinate, die alten Koordinaten benötigt werden.
Mit *Box()* wird der gesamte bisher gezeichnete Inhalt übermalt, da sonst die alten Linien immer noch sichtbar wären.

4 Objektorientierung

4.1 Installation von PureObject

Das Plugin PureObject ist für Windows und Linux verfügbar und stellt objektorientierte Programmierung als sogenannter Präprozessor zur Verfügung. Präprozessor heißt, das der objektorientierte Code, der standardmäßig nicht verfügbar ist, in normalen PureBasic-Code umgewandelt wird, damit der Compiler letztendlich den Code übersetzen kann.

Dieses Kapitel basiert auf der offiziellen Anleitung[1].

PureObject lädt man sich hier[2] herunter. Nachdem man das Paket entpackt hat, klickt man in der IDE auf *Werkzeuge-¿Werkzeuge* konfigurieren.... Hier legt man nun mit *Neu* einen neuen Eintrag an. Nun klickt man bei *Ereignis zum Auslösen des Werkzeugs* die Option *Vor dem kompilieren/Starten* an. Im rechten Kasten klickt man *Warten bis zum Beenden des Werkzeugs*, *Versteckt starten* und *Werkzeug vom Hauptmenü verstecken*. Im Eingabefeld Kommandozeile muss der Pfad zur *oop*-Datei angegeben werden, die man am Besten in das PureBasic-Verzeichnis kopiert. In das Feld *Argumente* schreibt man "*%FILE*" "*%COMPILEFILE*". Die Anführungsstriche sind hierbei mitzuschreiben. Bei Name kann man dem Werkzeug einen sinnvollen Namen geben, z.B. "OOP-Präprozessor 1".

Als nächstes legt man einen zweiten Eintrag an. Es werden genau die gleichen Dinge angegeben, mit der Ausnahme, dass bei *Ereignis zum Auslösen des Werkzeugs* die Option *Vor dem Erstellen des Executable* gewählt wird.

Unter *Datei-¿Einstellungen* sucht man den Eintrag *Eigene Schlüsselwörter*. Dort gibt man folgende Schlüsselwörter ein:

- Abstract
- Class
- DeleteObject
- EndClass
- Fixed
- Flex

[1] http://pb-oop.origo.ethz.ch/wiki/IDE_Installation
[2] http://pb-oop.origo.ethz.ch/wiki/pureobject

- Force
- NewObject
- This

Soviel zur Installation.

4.2 Einstieg in die Objektorientierte Programmierung

Die objektorientierte Programmierung basiert auf dem Paradigma der Objektorientierung. Ein Paradigma gibt an, in welchem Programmierstil der Programmierer an eine Problemstellung heran geht. PureBasic z.B. ist vorallem *imperativ* und *prozedual*, d.h. es werden Befehle in einer festen Reihenfolge verarbeitet und die Problemstellung wird in mehrere kleine Teilprobleme zerlegt (Stichwort *Procedure*), die für sich berarbeitet werden. Daraus ergibt sich die Lösung des gesamten Problems.
Bei der Objektorientierung versucht man das Problem in sogenannten Objekten darzustellen. Ein Objekt hat Eigenschaften (*Attribute*) und *Methoden*, um mit der Außenwelt, also anderen Objekten, in Kontakt zu treten. Die Attribute sind dabei von der Außenwelt abgeschnitten und die Methoden stellen Schnittstellen bereit, um auf diese zuzugreifen. Ähnliche Objekte werden in sogenannten *Klassen* zusammengefasst, in denen Attribute und Methoden deklariert werden. Man könnte also z.B. alle Menschen in einer Klasse zusammenfassen.

```
; Listing 36: Klassen

OpenConsole()

Class Mensch
  Mensch(alter.l = 0, name.s = "")
  Release()

  set_alter(alter.l)
  get_alter.l()
  set_name(name.s)
  get_name.s()

  alter.l
  name.s
EndClass

Procedure Mensch\Mensch(alter.l = 0, name.s = "")
  PrintN("Ich bin der Konstruktor")
  This\alter = alter
  This\name = name
```

```
EndProcedure

Procedure Mensch\Release()
  PrintN("Ich bin der Destruktor")
EndProcedure

Procedure Mensch\set_alter(alter.l)
  This\alter = alter
EndProcedure

Procedure.l Mensch\get_alter()
  ProcedureReturn This\alter
EndProcedure

Procedure Mensch\set_name(name.s)
  This\name = name
EndProcedure

Procedure.s Mensch\get_name()
  ProcedureReturn This\name
EndProcedure

*jesus.Mensch = NewObject Mensch(2011, "Jesus")

PrintN("Name: "+*jesus\get_name())
PrintN("Alter: "+Str(*jesus\get_alter()))

DeleteObject *jesus

Delay(2000)

Ausgabe:

Ich bin der Konstruktor
Name: Jesus
Alter: 2011
Ich bin der Destruktor
```

Ganz am Anfang wird die Klasse durch das Schlüsselwort *Class* deklariert. Die Konvention lauten, dass Bezeichner groß geschrieben werden sollten. Danach folgt die deklaration der Konstruktor- und Destruktor-Methode. Diese sind dazu da, um ein Objekt letztendlich zu initialisieren bzw. es wieder zu löschen. Dabei werden der Konstruktor-Methode zwei Argumente übergeben. Das = steht für einen Standardwert, falls kein Argument beim Methodenaufruf übergeben wird. Die Konstruktor-Methode hat dabei

immer den Namen der Klasse, die Destruktor-Methode heißt immer *Release()*.
Darauf folgen die Methoden, in diesem Fall Methoden, um auf die Attribute zuzugreifen. Attribute sind in PureBasic immer *privat*, d.h. für die Außenwelt nicht sichbar, in Gegensatz zu den Methoden, die *öffentlich* sind. Deshalb benötigt man diese *Setter* bzw. *Getter*. Zuletzt sei gesagt. dass Methoden immer außerhalb der Klasse definiert werden. Als letztes deklariert man die Attribute.
Bei der Methodendefinition schreibt man immer *Klassenname\Methodenname*. Der Backslash kann auch ein Punkt sein, ersterer fügt sich jedoch besser in die allgemeine PureBasic-Syntax ein. Weiter unten sieht man dann, wie auf Attribute zugegriffen wird: Die Attribute sind für die Methoden der gleichen Klasse sichtbar. Man schreibt *This\Attributname*. Auch hier gibt es wieder eine alternative Schreibweise: Anstatt des Backslash ein Pfeil (-¿) oder einfach nur der Pfeil, also auch ohne das *This*.
Ein Objekt wird mit dem Schlüsselwort *NewObject*, gefolgt vom Konstruktor erstellt, dem die Argumente für die Konstruktormethode übergegen werden können. Dabei wird eine Adresse auf das Objekt zurückgegeben, die in einem Zeiger vom Typ der Klasse gespeichert wird.
Die Methoden werden dann über den Zeiger aufgerufen, indem man schreibt **ptr\methode()*.
Zuletzt löscht man das Objekt wieder, wobei die Destruktor-Methode automatisch aufgerufen wird. Die Destruktor-Methode muss übrigens nicht definiert werden, wenn nichts beim Löschen des Objekts geschehen soll, in diesem Fall geschah dies nur zur Verdeutlichung.

4.3 Vererbung

Was tut man, wenn man einen Menschen hat, der Kung-Fu kämpfen kann, also etwas tun kann, dass nicht jeder Mensch kann. Jetzt könnte man natürlich eine komplett neue Klasse anlegen, dies ist aber ineffizient. Hier kommt *Vererbung* ins Spiel.
Der Kämpfer hat natürlich noch auch alle Eigenschaften eines normalen Menschens, also ein Alter und einen Namen. Man erstellt nun eine Klasse die von der ersten erbt.

```
; Listing 37: Vererbung

Class Mensch
  Mensch(alter.l = 0, name.s = "")
  Release()

  set_alter(alter.l)
  get_alter.l()
  set_name(name.s)
  get_name.s()

  alter.l
  name.s
EndClass
```

```
Procedure Mensch\Mensch(alter.l = 0, name.s = "")
  This\alter = alter
  This\name = name
EndProcedure

Procedure Mensch\set_alter(alter.l)
  This\alter = alter
EndProcedure

Procedure.l Mensch\get_alter()
  ProcedureReturn This\alter
EndProcedure

Procedure Mensch\set_name(name.s)
  This\name = name
EndProcedure

Procedure.s Mensch\get_name()
  ProcedureReturn This\name
EndProcedure

Class Kaempfer Extends Mensch
  Kaempfer()
  Release()

  kaempfen()
EndClass

Procedure Kaempfer\Kaempfer()
  This\Base(70, "Bruce")
EndProcedure

Procedure Kaempfer\kaempfen()
  PrintN("Hai Ya!")
EndProcedure


*bruce.Kaempfer = NewObject Kaempfer()

OpenConsole()

PrintN("Name: "+*bruce\get_name())
PrintN("Alter: "+Str(*bruce\get_alter()))
```

```
*bruce\kaempfen()

DeleteObject *bruce

Delay(2000)

Ausgabe:

Name: Bruce
Alter: 70
Hai Ya!
```

Die Vererbung wird durch das Schlüsselwort *Extend* eingeleitet, hinter das man die Elternklasse schreibt. Die Klassendeklaration funktioniert dann wieder auf die gleiche Weise.
Bei der Methodendefinition ist zu beachten, dass der Konstruktor den Konstruktor der Elternklasse aufrufen muss, da diesem Argumente übergeben werden sollen. Wenn der Konstruktor der Elternklasse Argumente ohne Standartwert erwartet, ist dieser Aufruf obligatorisch!
Man sieht, dass die Objektinitiliasierung ebenfalls gleich ist. Außerdem kann über den Zeiger auch auf die Methoden der Elternklasse zugegriffen werden. Vererbung ist also ein sinnvoller Weg, um viele Code-Zeilen zu sparen.

4.4 Polymorphismen

Es gibt Fälle, in denen soll eine erbende Klasse eine Methode neu definieren. Wenn man bei den vorhergegangen Beispiel der Menschen bleibt, könnte es passieren das der Ausgangsmensch Deutsch spricht, der neue jedoch Englisch. Man könnte nun auf die Idee kommen, die alte Methode einfach zu überschreiben.

```
; Listing 37: Polymorphismen

OpenConsole()

Class Mensch
  Mensch()
  Release()

  sprechen()
EndClass

Procedure Mensch\Mensch()
EndProcedure
```

```
Procedure Mensch\sprechen()
  PrintN("Hallo, ich bin ein Mensch")
EndProcedure

Class English Extends Mensch
  English()
  Release()

  sprechen()
EndClass

Procedure English\English()
EndProcedure

Procedure English\sprechen()
  PrintN("Hello, I'm a human being")
EndProcedure

*obj.English = NewObject English()

*obj\sprechen()

DeleteObject *obj

Delay(2000)

Ausgabe:

Hallo, ich bin ein Mensch
```

Wie man sieht, wird die Methode der Elternklasse aufgerufen, obwohl das Objekt von Typ *English* ist. Dieses Verhalten ist so gewollt. Möchte man jedoch, dass die Methode der erbenden Klasse aufgerufen wird, kommen nun die *Polymorphismen* ins Spiel. Polymorphie bedeutet, dass ein Ausdruck mit gleichem Bezeichner *kontextabhängig* interpretiert wird. Hier wurde durch den Bezeichner *sprechen()* die Methode der Elternklasse aufgerufen. PureObject stellt nun verschiedene Schlüsselwörter bereit, um dieses Verhalten zu ändern.

4.4.1 Flex und Force

Als erstes gibt es die Möglichkeit die Methode der Elternklasse *flexibel* zu gestalten, d.h. sie kann bei Bedarf von einer erbenden Klasse überschrieben werden.

```
; Listing 38: Flex
```

```
OpenConsole()

Class Mensch
  Mensch()
  Release()

  Flex sprechen()
EndClass

Procedure Mensch\Mensch()
EndProcedure

Procedure Mensch\sprechen()
  PrintN("Hallo, ich bin ein Mensch")
EndProcedure

Class English Extends Mensch
  English()
  Release()

  sprechen()
EndClass

Procedure English\English()
EndProcedure

Procedure English\sprechen()
  PrintN("Hello, I'm a human being")
EndProcedure

*obj.English = NewObject English()

*obj\sprechen()

DeleteObject *obj

Delay(2000)

Ausgabe:

Hello, I'm a human being
```

Durch das Schlüsselwort *Flex* vor der Methode *sprechen()* der Elternklasse wird dem Compiler gesagt, dass die Methode überschrieben werden kann.

```
; Listing 38: Flex

OpenConsole()

Class Mensch
  Mensch()
  Release()

  sprechen()
EndClass

Procedure Mensch\Mensch()
EndProcedure

Procedure Mensch\sprechen()
  PrintN("Hallo, ich bin ein Mensch")
EndProcedure

Class English Extends Mensch
  English()
  Release()

  Force sprechen()
EndClass

Procedure English\English()
EndProcedure

Procedure English\sprechen()
  PrintN("Hello, I'm a human being")
EndProcedure

*obj.English = NewObject English()

*obj\sprechen()

DeleteObject *obj

Delay(2000)
```

Ausgabe:

```
Hello, I'm a human being
```

Die zweite Methode sieht man hier: Durch das Schlüsselwort *Force* zwingt man den Compiler, die Elternmethode zu überschreiben. Würde man vor letztere das Schlüsselwort *Fixed* schreiben, würde *Force* einen Compilerfehler auslösen.

4.4.2 Abstrakte Methoden

Als letztes ist es möglich eine sogenannte *Abstrakte Klasse* zu schreiben. Bei dieser ist es obligatorisch, dass die Methoden überschrieben werden. Man kann sich die abstrakte Klasse wie ein Schema vorstellen, an dem man sich orientieren muss.

```
; Listing 38: Flex

OpenConsole()

Class Abstract Mensch
  sprechen()
EndClass

Class English Extends Mensch
  English()
  Release()

  sprechen()
EndClass

Procedure English\English()
EndProcedure

Procedure English\sprechen()
  PrintN("Hello, I'm a human being")
EndProcedure

*obj.English = NewObject English()

*obj\sprechen()

DeleteObject *obj

Delay(2000)

Ausgabe:

Hello, I'm a human being
```

Durch das Schlüsselwort *Abstract* wird ausgesagt, dass die nachfolgende Klassendeklaration abstrakt ist, d.h. jeder Mensch muss eine Methode *sprechen()* implemetieren. Die Klasse *English* erbt von dieser absrakten Klasse und muss folglich die Methode *sprechen()* erhalten und definieren.
Man beachte, dass die abstrakte Klasse keinen Konstruktor oder Destruktor hat, folglich kann man keine Objekte aus diesen Klassen erstellen. Auch werden die Methoden dieser Klasse nicht definiert.

5 Aufgabenlösungen

5.1 Grundlagen-Lösungen

5.1.1 Hello, world!

1. Hierfür gibt es keine Lösung, das liegt allein am Ehrgeiz des Programmiers.

2. Dies sind die Fehler im Listing:
   a) Es darf immer nur ein Befehl pro Zeile stehen.
   b) *PrintN()* erwartet einen String, es fehlen also die Anführungsstriche
   c) Hinter einem Befehl stehen immer geschlossene Klammern, *Delay()* fehlt also eine.

5.1.2 Variablentypen

1. Dies ist eine mögliche Lösung:

```
OpenConsole()
Print("Geben Sie den Radius des Kreises ein: ")
radius.f = ValF(Input())

PrintN("Der Umfang beträgt: "+StrF(2*radius*#Pi))

Delay(2000)
```

   Diese Aufgabe ist einfach: Es gibt eine Variable *radius* vom Typ Float, was durch das kleine f hinter dem Punkt dargestellt wird. Mit *ValF(Input())* kann man einen Radius eingeben, der direkt in einen Float umgewandelt wird. Der Umfang wird direkt in der Ausgabe berechnet, wobei Pi über eine PureBasic-interne Konstante dargestellt wird, die im Kapitel vorgestellt wurde.

5.1.3 Bedingungen

1. Dies ist eine mögliche Lösung:

```
OpenConsole()
Print("Geben Sie den Radius des Kreises ein: ")
radius.f = ValF(Input())
```

```
If radius < 0
 PrintN("Der Umfang beträgt: 0")
Else
 PrintN("Der Umfang beträgt: "+StrF(2*radius*#Pi))
EndIf

Delay(2000)
```

Die Aufgabe aus dem vorigen Kapitel wurde durch eine If-Else-Klausel erweitert, die überprüft, ob die Eingabe kleiner als 0 ist. Ansonsten arbeitet das Programm wie das erste.

2. Dies ist eine mögliche Lösung:

```
OpenConsole()
Print("Geben Sie die Geschwindigkeit des Autos ein: ")
geschwindigkeit.f = ValF(Input())
#Limit = 50

If geschwindigkeit <= 50
 PrintN("Keine Geschwindigkeitsübertretung.")
Else
 PrintN("Es werden "+StrF(#Limit-geschwindigkeit)+"\\
  zu schnell gefahren.")
EndIf

Delay(2000)
```

Die Aufgabe gleicht stark der ersten: Es wird die Geschwindigkeit abgefragt, danach wird noch eine Konstante angelegt, die das Geschwindigkeitslimit darstellt. Wenn die Geschwindigkeit nicht größer als 50 km/h ist, liegt keine Übertretung vor, ansonsten schon. Die Substraktion rechnet aus, wieviel zu schnell gefahren wurde.

5.1.4 Schleifen

1. Dies ist eine mögliche Lösung:

```
OpenConsole()
Print("Geben Sie einen String ein: ")
string.s = Input()
Print("Geben Sie eine ganze positive Zahl ein: ")
zahl = Val(Input())

If zahl < 1
```

```
  PrintN("Die Zahl muss größer als 0 sein!")
Else
  For k = 1 To zahl
    PrintN(string)
  Next
EndIf

Delay(2000)
```

Es wird ein zweidimensionales Array angelegt, das das Feld repräsentiert. In der Repeat-Schleife wird das Menü dargestellt und danach eine Eingabe erwartet. Auf diese wird dementsprechend reagiert: Bei 1 wird das Feld über eine verschachtelte For-Schleife ausgegeben, bei 2 müssen Koordinaten eingeben werden, die genutzt werden, um auf das Array zuzugreifen, und bei 3 bricht die Schleife ab.

5.1.5 Arrays, Listen und Maps

1. Dies ist eine mögliche Lösung:

```
OpenConsole()

Dim feld(4,4)

Repeat
  PrintN("1. Feld anzeigen")
  PrintN("2. Koordinate eingeben")
  PrintN("3. Programm beenden")
  Print("Auswahl: ")
  auswahl = Val(Input())

  Select auswahl
    Case 1
      For k = 0 To 4
        For l = 0 To 4
          Print(Str(feld(k,l)))
        Next
        PrintN("")
      Next
    Case 2
      Print("x-Koordinate: ")
      x = Val(Input())
      Print("y-Koordinate: ")
      y = Val(Input())
      feld(x-1,y-1) + 1
  EndSelect
```

```
  PrintN("")
Until auswahl = 3

Delay(2000)
```

Die Aufgabe kann einfach über eine For-Schleife gelöst werden, wobei die Anzahl der Iterationen vorher durch die Eingabe abgefragt wurde, ebenso wie der String, der ausgegeben werden soll. Die Anzahl der Iterationen muss dabei mindestens 1 sein.

5.2 Fortgeschrittene Themen-Lösungen

1. Eine mögliche Lösung ist:

```
OpenConsole()
Procedure ausgabe(Array zahlen.l(), mittel.l, laenge.l)
  For k = 0 To laenge - 1
    If zahlen(k) > mittel
      PrintN(Str(zahlen(k)))
    EndIf
  Next
EndProcedure

Dim a(10)
For k = 0 To 9
  a(k) = k
  summe+k
Next

ausgabe(a(), summe/10, 10)

Delay(2000)
```

Es wird ein Array mit 10 Elementen angelegt und in der For-Schleife mit den Zahlen von 1 bis 10 gefüllt. Gleichzeitig werden diese Zahlen aufaddiert. Dann wird die Prozedur aufgerufen, mit dem Array, dem Mittelwert und der Anzahl der Elemente des Arrays als Argumente. In der Prozedur wird dann das Array durchlaufen und überprüft, ob das aktuelle Element größer ist als der Mittelwert.

5.2.1 Prozeduren

5.3 Objektorientierung-Lösungen

6 Beispielprogramme

6.1 sFTPc

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  sFTPc - a simple FTP client
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
InitNetwork()

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  Structure to handle FTP-connections
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
Structure connection
  id.l
  adress.s
  user.s
  pass.s
  port.l
EndStructure

Declare establishConnection(List connections.connection())
Declare checkConnection(n.l)
Declare closeConnection(List connections.connection())
Declare fileManager(n.l)
Declare showDir(n.l)
Declare handleFileOptions(n.l)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  List for saving FTP-connections
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
NewList connections.connection()
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;   The menu
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
Procedure mainMenu(List connections.connection())
  Repeat

    ClearConsole()
    PrintN("sFTPc-Main Menu//")
    PrintN("1 - Open FTP Connection")
    PrintN("2 - Close FTP Connection")
    PrintN("3 - File-Manager")
    PrintN("4 - Quit")
    PrintN("")

    ;; Print opened connections
    If ListSize(connections()) > 0
      FirstElement(connections())
      ForEach connections()
        PrintN("Connection number: "+Str(connections()\id))
        PrintN("Adress: "+connections()\adress)
        PrintN("Username: "+connections()\user)
        Print("Connection: ")
        checkConnection(connections()\id)
        PrintN("")
      Next
    Else
      PrintN("There are currenty no opened connections.")
    EndIf

    Print("Choose: ")

    Select Val(Input())
      Case 1
        establishConnection(connections())
      Case 2
        closeConnection(connections())
      Case 3
        ClearConsole()
        Print("Connection number: ")
        n = Val(Input())
        If IsFTP(n)
          fileManager(n)
```

```
        Else
          PrintN("Not a valid connection!")
          Print("Push enter.")
          Input()
          Continue
        EndIf
      Case 4
        End
      Default
        PrintN("Not a valid option!")
        Print("Push enter.")
        Input()
        Continue
    EndSelect
  ForEver
EndProcedure

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  Procedure to open a new connection
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
Procedure establishConnection(List connections.connection())
  LastElement(connections())

  If Not ListSize(connections()) = 0
    lastID = connections()\id
  Else
    lastID = 0
  EndIf

  *newElement.connection = AddElement(connections())

  ClearConsole()
  *newElement\id = lastID + 1
  Print("URL or IP: ")
  *newElement\adress = Input()
  Print("Username: ")
  *newElement\user = Input()
  Print("Password: ")
  *newElement\pass = Input()
  Print("Port (default is 21): ")
  *newElement\port = Val(Input())
```

```
  If *newElement\port = 0
    *newElement\port = 21
  EndIf

  ;; Check if the connection can get opened
  If 0 = OpenFTP(*newElement\id,*newElement\adress,\\
  *newElement\user,*newElement\pass)
    PrintN("Could not open FTP-connection!")
    Print("Push enter.")
    Input()
    ;; Couldn't get opened? Delete the element, it's trash
    LastElement(connections())
    DeleteElement(connections())
  EndIf

  ProcedureReturn
EndProcedure

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  Check the state of opened connections
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
Procedure checkConnection(n.l)
  If 0 = CheckFTPConnection(n)
    PrintN("Disconnected!")
  Else
    PrintN("OK!")
  EndIf
  ProcedureReturn
EndProcedure

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  Close a specific connection
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
Procedure closeConnection(List connections.connection())
  ClearConsole()
  ;; Print current connections
  If ListSize(connections()) > 0
    FirstElement(connections())
    ForEach connections()
      PrintN("Connection number: "+Str(connections()\id))
```

```
      PrintN("Adress: "+connections()\adress)
      PrintN("Username: "+connections()\user)
      PrintN("")
    Next
  Else
    PrintN("There are currenty no opened connections.")
    Print("Push enter.")
    Input()
    ProcedureReturn
  EndIf

  Print("Choose Connection to close: ")
  n = Val(Input())

  If n <= 0 Or n > ListSize(connections()) Or (Not IsFTP(n))
    PrintN("Not a valid value!")
    Print("Push enter.")
    Input()
    ProcedureReturn
  EndIf

  CloseFTP(n)
  ForEach connections()
    If connections()\id = n
      Break
    EndIf
  Next

  DeleteElement(connections(),1)
  ProcedureReturn
EndProcedure

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  The build-in fileManager
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
Procedure fileManager(n.l)
  Repeat
    ClearConsole()

    showDir(n)

    PrintN("")
```

```
    PrintN("Type help for options.")
    Print("> ")

    If 1 = handleFileOptions(n)
      ProcedureReturn
    EndIf
  ForEver
EndProcedure

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  Show current directory
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
Procedure showDir(n.l)
PrintN("Current directory: "+GetFTPDirectory(n))
  ExamineFTPDirectory(n)

  ;; List files
  While NextFTPDirectoryEntry(n)
    attributes = FTPDirectoryEntryAttributes(n)
    If attributes & #PB_FTP_ReadUser
      Print("r")
    Else
      Print("-")
    EndIf
    If attributes & #PB_FTP_WriteUser
      Print("w")
    Else
      Print("-")
    EndIf
    If attributes & #PB_FTP_ExecuteUser
      Print("x")
    Else
      Print("-")
    EndIf
    If attributes & #PB_FTP_ReadGroup
      Print("r")
    Else
      Print("-")
    EndIf
    If attributes & #PB_FTP_WriteGroup
      Print("w")
    Else
```

```
      Print("-")
    EndIf
    If attributes & #PB_FTP_ExecuteGroup
      Print("x")
    Else
      Print("-")
    EndIf
    If attributes & #PB_FTP_ReadAll
      Print("r")
    Else
      Print("-")
    EndIf
    If attributes & #PB_FTP_WriteAll
      Print("w")
    Else
      Print("-")
    EndIf
    If attributes & #PB_FTP_ExecuteAll
      Print("x")
    Else
      Print("-")
    EndIf
    Print(" "+FTPDirectoryEntryName(n))
    If FTPDirectoryEntryType(n) = #PB_FTP_File
      PrintN(" File")
    ElseIf FTPDirectoryEntryType(n) = #PB_FTP_Directory
      PrintN(" Dir")
    Else
      PrintN("")
    EndIf
  Wend

  ProcedureReturn
EndProcedure

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  Handle the options of the fm
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
Procedure handleFileOptions(n.l)
  option.s = Input()
  Select option
    Case "changeDir"
```

```
    Print("Dir: ")
    newDir.s = Input()
    SetFTPDirectory(n, newDir)
    ProcedureReturn 0

  Case "createDir"
    Print("Name: ")
    name.s = Input()
    CreateFTPDirectory(n, name)
    ProcedureReturn 0


  Case "deleteDir"
    Print("Name: ")
    name.s = Input()
    DeleteFTPDirectory(n, name)
    ProcedureReturn 0


  Case "delete"
    Print("Name: ")
    name.s = Input()
    DeleteFTPFile(n, name)
    ProcedureReturn 0


  Case "download"
    Print("Name: ")
    name.s = Input()
    Print("Name for saving: ")
    saveName.s = Input()
    ReceiveFTPFile(n, name, saveName)
    ProcedureReturn 0


  Case "exit"
    ProcedureReturn 1


  Case "help"
    PrintN("changeDir -> change directory")
    PrintN("createDir -> create directory")
    PrintN("deleteDir -> delete EMPTY directory")
    PrintN("delete -> delete file")
```

```
      PrintN("download -> download file")
      PrintN("exit -> return to main menu")
      PrintN("help -> this help")
      PrintN("rename -> rename file or directory")
      PrintN("send -> send file to server")
      Print("Push enter.")
      Input()
      ProcedureReturn 0


    Case "rename"
      Print("Old Name: ")
      name.s = Input()
      Print("New name: ")
      newName.s = Input()
      RenameFTPFile(n, name, newName)
      ProcedureReturn 0


    Case "send"
      Print("File: ")
      name.s = Input()
      Print("Name for saving: ")
      saveName.s = Input()
      SendFTPFile(n, name, saveName)
      ProcedureReturn 0


    Default
      PrintN("Not a valid option")
      Print("Push enter.")
      Input()
      ProcedureReturn 0

  EndSelect
EndProcedure

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  Start the program
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
OpenConsole()
EnableGraphicalConsole(1)
```

```
mainMenu(connections())
```

6.1.1 Was soll das Programm tun?

Dieses Programm stellt einen einfachen, konsolenbasierten FTP-Client dar. Wenn das Programm gestartet wird, sieht man im Terminal ein Menü. Über den ersten Menüpunkt kann man eine FTP-Session eröffnen, über den zweiten kann man sie wieder schließen und über den dritten kann man die Dateien auf dem FTP-Server durchsuchen und bearbeiten. Der vierte schließt das Programm wieder. Das Programm ist dabei in der Lage mehrere FTP-Sessions zu verwalten. Man muss immer angeben, welche Session gerade bearbeitet werden soll.

6.1.2 Wie funktioniert das Programm?

Am Anfang wird über *InitNetwork()* die Netzwerk-Bibliothek geladen, wodurch die Befehle für die FTP-Verwaltung verfügbar werden. Die Struktur ist da, um die FTP-Verbindungen zu verwalten. Danach folgt die Deklaration mehrerer Verbindungen, sowie die Deklaration einer Liste, in der die FTP-Verbindungen, repräsentiert über die Struktur *connection*, orgranisiert werden. Danach werden die Prozeduren definiert. Ganz am Ende des Programmes wird die Konsole geöffnet. *EnableGraphicalConsole(1)* ermöglicht das Löschen des Inhalts der Konsole. Der letzte Befehl ruft das Hauptmenü auf, dass in der ersten Prozedur definiert ist.

- *mainMenu()*: Die Prozedur erwartet, dass man ihr die Liste mit den Verbindungen übergibt. Es wird lediglich das Menü dargestellt und auf Eingaben mittels Fallunterscheidungen reagiert. Wenn FTP-Verbindungen eröffnet wurden, stellt das Programm diese dar. Beim Schließen des Programms muss nicht darauf geachtet werden, dass alle Verbindungen geschlossen werden, dies wird automatisch beim Schließen erledigt.

- *establishConnection()*: Diese Prozedur erwartet ebenfalls die Liste der Verbindungen. Zuerst wird die Nummer der letzten FTP-Verbindung herausgefunden. Das ist wichtig für die Vergabe einer Nummer an die neue Verbindung. Danach wird ein neues Element zur Liste hinzugefügt. *AddElement()* gibt die Adresse des neuen Elements zurück, die im Zeiger gespeichert wird. Über den Zeiger wird danach auf das Element zugegriffen und die Daten werden eingegeben. Mit *OpenFTP()* wird dann versucht die Verbindung herzustellen. Wenn sie nicht hergestellt werden kann, wird sie gelöscht.

- *checkConnection()*: Diese Prozedur überprüft lediglich, ob die Verbindung noch besteht. Dies wird im Hauptmenü genutzt, wo dies angezeigt wird.

- *closeConnection()*: Es werden alle geöffneten Verbindungen angezeigt. Man wählt eine aus und diese wird geschlossen und aus der Liste gelöscht. Es wird dabei überprüft, ob es sich wirklich um eine geöffnete Verbindung handelt.

- *fileManager()*: Diese Prozedur steuert den Dateimanager. Es wird das aktuell ausgewählte Verzeichnis angezeigt und man kann mit diesem interagieren.
- *showDir()*: Es wird einfach der Inhalt des Verzeichnis aufgelistet, zusammen mit den Rechten des Benutzers, mit dem die Verbindung eröffnet wurde.
- *handleFileOptions()*: Diese Prozedur wird aus *fileManager()* aufgerufen. Man kann verschiedene Befehle eingeben. Welche das sind, wird über den Befehl *help* angezeigt.

6.2 reversePolishNotation

```
OpenConsole()

NewList entries.s()
PrintN("Please type only numbers and arithmetic operators(+,-,*,/)")
PrintN("Input:")

Repeat
  entry.s = Input()
  AddElement(entries())
  entries() = entry

  If entries() = "="
    DeleteElement(entries())
    PrintN(entries())
    Input()
    Break
  EndIf
  If entries() = "-" Or entries() = "+" Or entries() = "/" Or\\
   entries() = "*"
    If ListSize(entries()) < 3
      PrintN("Not enough Elements for calculating")
      Input()
      Break
    EndIf

    index = ListIndex(entries())
    SelectElement(entries(),index-2)

    a.f = ValF(entries())
    DeleteElement(entries())
    NextElement(entries())
```

```
      b.f = ValF(entries())
      DeleteElement(entries())
      NextElement(entries())

      Select entries()
        Case "-"
          InsertElement(entries())
          entries() = StrF(a-b)
          NextElement(entries())
          DeleteElement(entries())
        Case "+"
          InsertElement(entries())
          entries() = StrF(a+b)
          NextElement(entries())
          DeleteElement(entries())
        Case "/"
          InsertElement(entries())
          entries() = StrF(a/b)
          NextElement(entries())
          DeleteElement(entries())
        Case "*"
          InsertElement(entries())
          entries() = StrF(a*b)
          NextElement(entries())
          DeleteElement(entries())
      EndSelect
    EndIf
  ForEver
```

6.2.1 Wie funktioniert das Programm?

Dieses Programm stellt einen Taschenrechner dar, jedoch einen besonderen: Er arbeitet mit der umgekehrten polnischen Notation. Hier schreibt man die zwei Zahlen, mit denen gerechnet werden soll zuerst und danach die Rechenoperation. "2+8"würde man also als "2 8 +ßchreiben. Hierdurch kann man sich außerdem die Klammersetzung sparen: "2*(5-3)"könnte man schreiben als "2 5 3 - *". Diese Notation basiert also auf einer einfachen Stapelverarbeitung, d.h. man legt alle Zahlen metaphorisch auf einen Stapel und sobald man eine Rechenoperation herauflegt, wird mit den beiden Zahlen, die darunter liegen, gerechnet, je nachdem welche Operation heraufgelegt wurde. Die berechnete Zahl wird dann auf den Stapel gelegt, nachdem die Rechenoperation und die beiden Zahlen, mit denen gerechnet wurde, gelöscht wurden. Diese Beschreibung wurde hier 1:1 implementiert. Der Stapel wurde jedoch nicht implementiert, stattdessen wird eine Liste verwendet, mit der dynamisch gearbeitet wird.